# foo_rpc Documentation

***Release 0.1***

**Claudiu Popa**

**Nov 05, 2016**

Contents

**foo_rpc** is a foobar2000 component which exports an API access to some of the internal services of the audio player. This is similar in nature with foo_com_automation, although there are differences in the approach of both projects.

Contents:

CHAPTER 1

# Implementation

The component is based on two technologies:

- instead of relying on COM, **foo_rpc** uses Windows named pipe for the communication protocol, having the address **\\.\pipe\foobar2000**.
- the parameters to the API methods, as well as the responses are serialized with msgpack.

The APIs are exported as qualified names, where the first part is the class to which the given method belongs and the second part is the actual method itself. For instance, for starting the playback, one can call the method `PlaybackControl.start`, while for sorting the current playlist by a specific criteria, one can use `Playlist.activeplaylist_sort_by_format`. For more informations, see *foo_rpc API methods*.

Due to implementation limitations, we cannot know a priori what parameters a method requires when dispatching the method. Due to this, the components needs to receive a two-packed object, containing the method to be called and its parameters. For example, to call the **start** method, one could call the API with this payload:

```
msgpack::sbuffer sbuf;
msgpack::pack(sbuf, std::make_tuple(0, false));
auto params = sbuf.data();

msgpack::sbuffer sbuf;
msgpack::pack(sbuf, std::make_tuple("PlaybackControl.start", params));
auto payload = sbuf.data();
```

In this case, the component will know the first element of the tuple is the method, while the second is the parameters. In case the method does not need any parameters, the second element will not be unpacked, meaning that the first packing is not necessary in this case.

Each API call returns a two-element tuple, where the first element designates the status of the request, while the second one represents the result or the error, in case the request failed. In case the request succeeds, the first element is 0.

CHAPTER 2

# foo_rpc API methods

This section contains a thorough description of each method that **foo_rpc** accepts. The parameters of the methods are expected to be packed with msgpack.

`CoreVersion.`**`get_name`**`()`

**Returns** string

Return the name of the program, e.g. "foobar2000".

`CoreVersion.`**`get_version_as_text`**`()`

**Returns** string

Return the version string of the program, in the form "N.N.N.N"

`PlaylistLoader.`**`load_playlist`**`(`*string playlist*`)`

**Returns** bool

Attempts to load a playlist file from specified filesystem path.

`PlaylistLoader.`**`save_playlist`**`(`*string playlist_path*, *vector<string> files*`)`

**Returns** bool

Saves specified list of locations into a playlist file.

`PlaybackControl.`**`get_now_playing`**`()`

**Returns** tuple<bool, optional<Track>>

Retrieves now playing item handle. Returns true on success, false on failure (not playing).

`PlaybackControl.`**`start`**`(`*tuple<play_control::t_track_command, bool>*`)`

Starts playback. If playback is already active, existing process is stopped first.

The first parameter specifies what track to start playback from. See t_track_Command enum from foobar2000's SDK for more info.

The second parameter specifies whether playback should be started as paused.

`PlaybackControl.`**`stop`**`()`

Stops playback.

`PlaybackControl.`**`is_playing`**`()`

**Returns** bool

Returns whether playback is active.

`PlaybackControl.`**`is_paused`**`()`

**Returns** bool

Returns whether playback is active and in paused state.

PlaybackControl.**pause**(*bool p_state*)
Toggles pause state if playback is active.

**Parameters** **p_state** – set to true when pausing or to false when unpausing.

PlaybackControl.**get_stop_after_current**()

**Returns** bool

Retrieves stop-after-current-track option state.

PlaybackControl.**set_stop_after_current**(*bool p_state*)
Alters stop-after-current-track option state.

PlaybackControl.**set_volume**(*float p_value*)
Alters playback volume level. :param p_value: volume in dB; 0 for full volume.

PlaybackControl.**get_volume**()

**Returns** float

Retrieves playback volume level. Returns current playback volume level, in dB; 0 for full volume.

PlaybackControl.**volume_up**()
Alters playback volume level one step up.

PlaybackControl.**volume_down**()
Alters playback volume level one step down.

PlaybackControl.**volume_mute_toggle**()
Toggles playback mute state.

PlaybackControl.**playback_seek**(*double p_time*)
Seeks in currenly played track to specified time. :param p_time: target time in seconds.

PlaybackControl.**playback_seek_delta**(*double p_delta*)
Seeks in currently played track by specified time forward or back. :param p_delta: time in seconds to seek by; can be positive to seek forward or negative to seek back.

PlaybackControl.**playback_can_seek**()

**Returns** bool

Returns whether currently played track is seekable. If it's not, playback_seek/playback_seek_delta calls will be ignored.

PlaybackControl.**playback_get_position**()

**Returns** double

Returns current playback position within currently played track, in seconds.

PlaybackControl.**playback_format_title**(*string script*)

**Returns** string

Renders information about currently playing item. The script represents the query to use for formatting the current playing item.

PlaybackControl.**playback_format_title_complete**()

**Returns** string

Similar to *PlaybackControl.playback_format_title()*, with the caveat that this function has the script hardcoded.

PlaybackControl.**playback_get_length**()

**Returns** double

Retrieves length of currently playing item.

PlaybackControl.**playback_get_length_ex**()

**Returns** double

Extended version: queries dynamic track info for the rare cases where that is different from static info.

PlaybackControl.**toggle_stop_after_current**()
Toggles stop-after-current state.

PlaybackControl.**toggle_pause**()
Toggles pause state.

PlaybackControl.**play_or_pause**()
Starts playback if playback is inactive, otherwise toggles pause.

PlaybackControl.**play_or_unpause**()

PlaybackControl.**next**()

PlaybackControl.**previous**()

PlaybackControl.**is_muted**()

**Returns** bool

Check if the volume is muted.

PlaybackControl.**get_volume_step**()

**Returns** double

Returns user-specified the step dB value for volume decrement/increment.

Playlist.**get_playlist_count**()

**Returns** t_size

Returns number of playlists.

Playlist.**get_active_playlist**()

**Returns** t_size

Retrieves index of active playlist; infinite if no playlist is active.

Playlist.**set_active_playlist**(*t_size*)
Sets active playlist (infinite to set no active playlist)

Playlist.**get_playing_playlist**()

**Returns** t_size

Retrieves playlist from which items to be played are taken from.

Playlist.**set_playing_playlist**(*t_size*)
Sets playlist from which items to be played are taken from.

Playlist.**remove_playlists**(*vector<t_size>*)
Removes playlists with the ids from the given list.

Playlist.**create_playlist**(*string name*, *t_size length*, *t_size index*)

**Returns** size_t

Creates a playlist with the given name. Do note that the API needs the length of the name. For instance, this might be useful in case the name is UTF-8, in which case the actual length might be double than what is expected.

Playlist.**create_playlist_ex**(*string name*, *t_size length*, *t_size index*, *vector<string> files*)

**Returns** size_t

Creates a playlist with the given name. Similar notes to *Playlist.create_playlist()* applies. The **files** parameter contains the files that are added to the initial content of the playlist.

Playlist.**reorder**(*vector<int> permutations*)

**Returns** bool

Reorder the playlists according to the given permutations.

Playlist.**playlist_get_item_count**(*t_size playlist*)

**Returns** size_t

Get the number of items from the given playlist.

Playlist.**activeplaylist_get_item_count**()

**Returns** size_t

Get the number of items from the active playlist.

Playlist.**playlist_get_name**(*t_size playlist*)

**Returns** string

Get the playlist's name.

Playlist.**activeplaylist_get_name**()

**Returns** string

Get the name of the active playlist.

Playlist.**playlist_reorder_items**(*t_size playlist*, *vector<int> permutation*)

**Returns** bool

Reorder the items in the specified playlist according to the specified permutation.

Playlist.**activeplaylist_reorder_items**(*vector<int> permutation*)

**Returns** bool

Reorder the items in the active playlist according to the specified permutation.

Playlist.**playlist_remove_items**(*t_size playlist*, *vector<t_size> items*)

**Returns** bool

Remove the specified items from the given playlist.

Playlist.**activeplaylist_remove_items**(*vector<t_size> items*)

**Returns** bool

Remove the specified items from the active playlist.

Playlist.**playlist_replace_item**(*t_size playlist*, *t_size item*, *t_size replacee*)

**Returns** bool

Replace the given item with the one specified by the **replacee** parameter, in the given playlist.

`Playlist.`**`activeplaylist_replace_item`**(*t_size item*, *t_size replacee*)

**Returns** bool

Replace the given item with the one specified by the **replacee** parameter, in the active playlist.

`Playlist.`**`playlist_insert_items`**(*t_size playlist*, *t_size base*, *vector<string> files*)

**Returns** t_size

Insert the given files at the specified position, in the given playlist.

`Playlist.`**`activeplaylist_insert_items`**(*t_size base*, *vector<string> files*)

**Returns** t_size

Insert the given files at the specified position, in the active playlist.

`Playlist.`**`playlist_rename`**(*t_size playlist*, *std::string name*, *t_size length*)

**Returns** bool

Change the name of the given playlist.

`Playlist.`**`activeplaylist_rename`**(*std::string name*, *t_size length*)

**Returns** bool

Change the name of the active playlist.

`Playlist.`**`playlist_activate_next`**()
Activate the next playlist.

`Playlist.`**`playlist_activate_previous`**()
Activate the previous playlist.

`Playlist.`**`playlist_add_items`**(*t_size playlist*, *vector<string> files*)

**Returns** t_size

Add the given files in the given playlist.

`Playlist.`**`activeplaylist_add_items`**(*vector<string> files*)

**Returns** t_size

Add the given files in the active playlist.

`Playlist.`**`playlist_item_format_title`**(*t_size playlist*, *t_size item*, *string format*)

**Returns** string

Format the given item with the given format string. The item is retrieved from the given playlist.

`Playlist.`**`activeplaylist_item_format_title`**(*t_size item*, *string format*)

**Returns** string

Format the given item with the given format string. The item is retrieved from the active playlist.

`Playlist.`**`get_playing_item_location`**()

**Returns** std::tuple<bool, t_size, t_size>

Get the playing item location. It returns a tuple of three elements, where the first element is a bool representing if there is a playing item or not, the second item represents the playlist where the playing item is located and the last element is the playing item's position in the given playlist.

Playlist.**playlist_sort_by_format**(*t_size playlist*, *string format*, *bool sel_only*)

**Returns** bool

Sort the given playlist by the given format string. The last parameter controls if the sorting should be done on the selection only or not.

Playlist.**activeplaylist_sort_by_format**(*string format*, *bool sel_only*)

**Returns** bool

Sort the active playlist by the given format string. The last parameter controls if the sorting should be done on the selection only or not.

Playlist.**playback_order_get_count**()

**Returns** t_size

Get the number of the playback items.

Playlist.**playback_order_get_name**(*t_size item*)

**Returns** string

Get the name of the given playback item.

Playlist.**playback_order_get_active**()

**Returns** t_size

Get the index of the active playback order item.

Playlist.**playback_order_set_active**(*t_size item*)
Set the given playback order item as active.

Playlist.**queue_add_item_playlist**(*t_size playlist*, *t_size p_item*)
Add the given item, from the given playlist, into the playback queue.

Playlist.**queue_get_count**()

**Returns** t_size

Get the number of items in the playback queue.

Playlist.**queue_get_contents**()

**Returns** std::vector<std::map<std::string, t_size>>

Return the list of playback queue's items. A playback queue's item is represented as a dictionary, containing the song's position and its playlist.

Playlist.**queue_remove_mask**(*vector<t_size> items*)
Remove the given items from the playback queue.

Playlist.**queue_flush**()
Remove all the items from the playback queue.

Playlist.**queue_is_active**()

**Returns** bool

Check if the playback queue is active.

Playlist.**remove_playlist**(*t_size playlist*)

**Returns** bool

Remove the given playlist.

`Playlist.`**`remove_playlist_switch`**(*t_size playlist*)

**Returns** bool

Remove the given playlist and the switch to the next one.

`Playlist.`**`playlist_clear`**(*t_size playlist*)
Clear the given playlist.

`Playlist.`**`activeplaylist_clear`**()
Clear the active playlist.

`Playlist.`**`create_playlist_autoname`**(*t_size position.*)

**Returns** t_size

Create a playlist with a default name at the specified position.

`Playlist.`**`reset_playlist_playlist`**()

`Playlist.`**`find_playlist`**(*string name*, *t_size length*)

**Returns** t_size

Find the playlist with the given name.

`Playlist.`**`find_or_create_playlist`**(*string name*, *t_size length*)

**Returns** t_size

Find the playlist with the given name. If it does not exist, it is going to be created with the given name.

`Playlist.`**`get_all_items`**(*t_size playlist*)

**Returns** std::vector<Track>

Get all the items from the given playlist. Returns a list of tracks, where a track is a tuple of pairs, as in:

```
(
    ("index", song_index),
    ("path", song_path),
    ("selected", is_song_selected),
    ("subsong_index", subsong_index)
)
```

`Playlist.`**`activeplaylist_get_all_items`**()

**Returns** std::vector<Track>

See the documentation of *`Playlist.get_all_items()`*

`Playlist.`**`playlist_get_item_handle`**(*t_size playlist*, *t_size item*)

**Returns** optional<Track>

Get the handle of the given item, from the given playlist. If the item does not exist, this function will return a null. Otherwise, it will return a Track object, which is a tuple of pairs, with the format:

```
(
    ("index", song_index),
    ("path", song_path),
    ("selected", is_song_selected),
```

```
    ("subsong_index", subsong_index)
)
```

Playlist.**activeplaylist_get_item_handle**(*t_size item*)

**Returns** optional<Track>

Get the handle of the given item, from the active playlist.

Playlist.**playlist_get_items**(*t_size playlist*, *vector<t_size> masks*)

**Returns** std::vector<Track>

Get the items specified by **masks** from the given playlist.

Playlist.**activeplaylist_get_items**(*vector<t_size> masks*)

**Returns** std::vector<Track>

Get the items specified by **masks** from the active playlist.

CHAPTER 3

# Indices and tables

- genindex
- modindex
- search

## C



## P